SLAAC1-V SDK USER'S MANUAL

Release 0.3.1

Peter Bellows

October 1, 2000

Table of Contents

1 Introduction....3
2 Setup....3
2.1 Windows NT (Visual C++) setup....3
2.2 Linux setup....4
3 Using the Slaac1VBoard API....4
3.1 The Slaac1VBoard class....4
3.2 Configuring the PEs....5
3.2.1 Configuration cache control....6
3.2.2 Partial configuration (advanced users ONLY!!!)....7
3.3 Setting Up the FPGA Clock (PCLK)....7
3.3.1 Setting basic clock modes....7
3.3.2 "FIFO-step" mode....8
3.3.3 PE Stop-clock mode....8
3.4 Controlling the FIFOs....8
3.4.1 Low-overhead Enqueue/Dequeue equivalents....8
3.4.2 Accessing the FIFO tags....9
3.5 Accessing the PE Memories....9
3.5.1 32-bit mode vs. 36-bit mode....10
3.5.2 Preemptive vs. non-preemptive....10
3.5.3 DMA vs. PCI slave mode and pipelined vs. flow-through mode 11
3.6 Performing PE Readback....11
3.7 FPGA Control....12
3.8 Reprogramming the Configuration Controller ("It's not a bug, it's a feature!")....12
3.9 Interrupt Control....12
3.10 Slaac1VBoard Register Access....14

1 Introduction

Purpose:
This manual describes how to use the C++ API for controlling the SLAAC1-V board, and then describes the basic functionality of the key circuit modules in IF. It does **not** describe the following:

- SLAAC1-V general architecture and VHDL simulation environment; this is documented in the "SLAAC1-V User VHDL Guide"
- SLAAC1-V register-level definitions; this is described in "SLAAC1-V Register Definitions"
- The full reference for the Slaac1VBoard API. This is contained in the "Slaac1VBoard API Reference"

2 Setup

****IMPORTANT**:** The Slaac1VBoard classes can no longer be compiled independently of the Virginia Tech ACS_API . You must obtain a copy of that distribution. This release of the Slaac1VBoard classes was compiled against version 1.3.0 of ACS_API. It is highly recommended that you program directly with that API where possible, as it will allow the easiest path for porting to other ACS systems as well as the easiest path for supporting more complex multi-board environments.

2.1 Windows NT (Visual C++) setup

As best we can tell, Visual C++ libraries are not forward-compatible; this means that you have to be using the same version (6.0) as we are.

To use the SLAAC1-V SDK with Visual C++ 6.0, you need to do three things (this is already done for you in the SampleApp project):

a. Set up your include path to point to all of the header files in the distribution. Go to Project->Settings, and click on the "C++" tab. Select the "Preprocessor" category on that pane, and add <SLAAC1V-ROOT>\include and <ACS_API-ROOT>\include to the "Additional include directories field".
b. Add the SLAAC1-V libraries to the compilation. Go to Project->Settings, and click "Link" tab. Add all files in <SLAAC1V-ROOT>\lib to the "Object/library modules" field. Also add all pertinent libraries from <ACS_API-ROOT>\lib to that field (i.e. slaac_api_WIN32.lib).
c. Put '#include <Slaac1VBoard.h>' in your file. Click! Should compile very nicely. Lately I have been getting a linker warning about the LIBCMTD library, which is not a problem.

2.2 Linux setup

a. Add <SLAAC1V-ROOT>/include and <ACS_API-ROOT>/include to your preprocessor include path: "-I$SLAAC1V/include –I$ACS_API/include", for example.
b. Add all the ".a" files in <SLAAC1V-ROOT>/lib to the linker step (i.e. the final c-compiler call which produces the executable). Also add all pertinent ".a" files from <ACS_API-ROOT>/lib to the linker step (i.e. slaac_api_LINUX.a). Also, the Linux distribution depends on the "curses" package for some of its message reporting; make sure to add "-lcurses" to your linker step as well.
c. Put '#define LINUX' followed by #include <Slaac1VBoard.h>' in your file. Click! Should compile very nicely. Of course you can also add '-DLINUX" to your compiler invocations instead of defining it in the source, if that's not already done by default.

General things to note about the Slaac1VBoard class:

- The SLAAC1-V DK distributions for Linux and Windows NT currently come from an identical code base. Therefore you should be able to cross-compile your source code to either OS, as long as you link in libraries from the same release.
- Slaac1VBoard->InitBoard(...) is the safest way to configure the board for new users. If you want to do a custom initialization step, it is strongly recommended that you study the comments for that method to understand what kinds of things you need to do for a safe initialization.
- Configuring / deconfiguring X0 affects the state of the FIFOs; see the comments.
- Read / Write / Enqueue / Dequeue all use byte-counts (for compatibility with VTU's API), but always expect to operate on 32-bit words (memory access) or 64-bit words (FIFO access).
- Default FIFO accesses are 'blocking'. If you want, you can use EnqueueNonBlocking(...) or DequeueNonBlocking(...) for non-blocking accesses.
- In some cases, some of the board functionality that we wanted to bring out is not currently covered under the VTU API. Therefore Slaac1VBoard has a lot of methods that are currently unique to SLAAC1-V. While this gives you maximum control over board operations, using these "non-core" methods means that your code will not be directly portable to other platforms targeted by the general SLAAC API in the future.

3 Using the Slaac1VBoard API

3.1 The Slaac1VBoard class

All hardware interaction is encapsulated in the C++ class **Slaac1VBoard**. This class exports methods to perform all board-level functions, such as configuring PEs, reading/writing memories, reading/writing FIFOs, setting clocks, etc. To use the

SLAAC1-V libraries, you simply need to point to the include files and libraries indicated in the setup section. The SDK distribution comes with a sample Visual C++ 6.0 workspace with these project settings already made.

To begin using the board, you instantiate a Slaac1VBoard object:

```
Slaac1VBoard *s1board = new Slaac1VBoard(); // opens the board
or:
// exclusively opens the board
Slaac1VBoard *s1board = new Slaac1VBoard(TRUE);
```

The difference between the two invocations is that the latter invocation will guarantee that no other processes can access the board until this process releases it. Optionally, you can also specify a board number, 0 … N-1, if you have multiple SLAAC1-V boards installed in your host, as follows:

```
// opens board #1 non-exclusively (only works if you have at
// least 2 SLAAC1-V boards installed in your host.
Slaac1Vboard *s1board = new Slaac1Vboard(1, TRUE);
```

The Slaac1VBoard inherits from the **SlaacBoard** class, which encapsulates operations that are common between the different SLAAC architectures. SlaacBoard in turn inherits from **ACS_CLocalNode** from Virginia Tech's ACS_API library. The Slaac1VBoard has a bunch of SlaacRegister objects, each of which represents a specific control register on the board (most of which are inherited from SlaacBoard). You **should not have to access these registers directly, and can really mess up your board if you play with these without knowing what you are doing.** All user-level functionality should be encapsulated into the Slaac1VBoard public methods; however these registers are exposed in the event that the existing API just isn't quite meeting your needs and you need to customize a control function. As an example: the member Slaac1VBoard::HDSKX0 controls the handshake bus between IF and X0, but in general you really only need to access the Slaac1Board->SetHandshake/GetHandshake(…) methods to handle all your handshake control. Many of these methods are defined by the top-level ACS API, which has rich communications and remote programming constructs for heterogeneous systems. Please refer to VT's ACS API documentation for more details of the top-level API.

From this point on, we describe the basic kinds of functionality that is available to the SLAAC1-V user through the Slaac1VBoard methods.

3.2 Configuring the PEs

Configuration is done through two Slaac1VBoard methods (from here on we assume that *s1board* is declared as *Slaac1VBoard *s1board*). First, we can manually load the bitstream file, and then pass the bitstream to Slaac1VBoard->Configure as follows:

```
ACS_CONFIG config;
config.pe_mask = SLAAC_X0; // or SLAAC_X1 or SLAAC_X2
s1board->LoadConfigurationFile(filename, &config);
```

```
s1board->Configure(&config);
free(config.bitstream); // Remember to do this, as it is
                        // malloc'ed by Load…!
```

A safer way is to use the InitBoard method, which configures all PEs, and then brings up the board in a safe, known state:

```
char *x0_bitfile, *x1_bitfile, *x2_bitfile…
s1board->InitBoard(x0_bitfile, x1_bitfile, x2_bitfile,
      40.0 /* MHz */);
```

The InitBoard method guarantees that:

1. The PEs have been configured.
2. Global reset (GSR) has been asserted and then deasserted
3. The clock has been set to the specified frequency
4. FIFOs have been reset
5. Handshake lines have been tri-stated
6. Interrupts are disabled during initialization, and then restored after initialization.

So the recommended way to configure the board is through InitBoard, because of its conservative boot sequence. If you need a custom configuration routine, you can still use *InitBoard(double clock_freq)* or *InitBoard()* to run the initialization sequence without configuring the PEs.

3.2.1 Configuration cache control

The SLAAC1-V architecture includes rich configuration cache support. The on-board configuration control has enough SRAM to store 4 entire XCV1000 bitstreams, plus enough flash RAM to store 4 more bitstreams. All X0 bitstreams must be loaded into one of these cache locations before they can be downloaded to X0. By default, all X0 configurations are stored in SRAM cache location #0. However, the user can control this cache behavior using the ConfigureCache method. For example:

```
ACS_CONFIG config;
config.pe_mask = SLAAC_X0;
s1board->LoadConfigurationFile(filename, &config);
s1board->ConfigureCache(&config, Flash, 1);
```

would cause the configuration to be cached in location #1 in flash RAM, then downloaded. ***Think really hard before you write something to flash location #0, as this is the default power-up PCI interface; overwriting this with a bad bitstream may render your board inoperable (until you can patch in a correct bitstream via download cable)***. As another example, the following shows how to configure X0 with the bitstream that is already in SRAM cache location #3:

```
ACS_CONFIG config;
config.pe_mask = SLAAC_X0;
config.bitstream = NULL; // Setting this to NULL means use the
                         // previously-cached bitstream
```

```
s1board->ConfigureCache(&config, Sram, 3);
```

In the future, this method will also support configuring X1 and X2 from the bitstream cache, as this feature becomes available in hardware.

3.2.2 Partial configuration (advanced users ONLY!!!)

Partial configuration can be performed using the *ConfigurePartial* method as shown below. From the control API point of view, the only difference between regular configuration and partial configuration is that we do not clear the device (by asserting PROGRAM_N) when we partially configure. Please note that generating a correct partial bitstream requires a lot of care.

```
ACS_CONFIG config;
config.pe_mask = SLAAC_X0; // or SLAAC_X1 or SLAAC_X2
s1board->LoadConfigurationFile(filename, &config);
s1board->ConfigurePartial(&config, TRUE);
free(config.bitstream); // Remember to do this, as it is
                        // malloc'ed by Load…!
```

3.3 Setting Up the FPGA Clock (PCLK)

The SLAAC1-V board has a very flexible clocking scheme, with a programmable clock synthesizer and several clocking modes:

- Free-running
- Step mode (arbitrary number of steps)
- FIFO-step mode (a FIFO write stimulates a set number of clock steps)
- PE-stop-clock mode

The clock frequency is set using the Clock_Set method:

```
ACS_CLOCK clock;
clock.frequency = 40.0;
clock.countdown = 0;
s1board->Clock_Set(&clock);
```

If we set *clock.countdown* to a non-zero value, the Clock_Set method runs that number of clock steps. More flexible and complete clock control is provided by the methods described below.

3.3.1 Setting basic clock modes

We can use the following methods to set the clock mode:

```
s1board->Run(); // starts PCLK in free-run mode
s1board->Run(40.0); // starts PCLK in free-run mode at 40 MHz
s1board->Stop(); // stops PCLK
s1board->Step(count);   // Starts a step sequence and waits for
                        // it to finish
s1board->Run(count);    // Starts a step sequence but does not
                        // wait for it to finish
```

3.3.2 "FIFO-step" mode

We can set up "FIFO-step" mode as follows:

```
s1board->SetFifoStepMode(TRUE, 5);
s1board->WriteFifo(data); // Will write data, then issue 5 PCLKs
s1board->SetFifoStepMode(TRUE, 20);
s1board->WriteFifo(data); // Will write data, then issue 20 PCLKs
s1board->WriteFifo(data); // Will write data, then issue 20 PCLKs
s1board->WriteFifo(data); // Will write data, then issue 20 PCLKs
```

3.3.3 PE Stop-clock mode

We can set up "PE-stop-clock" mode as follows. A PE halt is signaled by a high-to-low transition on the corresponding X#_HALT signal. To enable this functionality, we do the following:

```
s1board->EnablePEHalt(SLAAC_X1, TRUE);
```

After doing this, a high-to-low transition on X1_HALT will stop the clock. By definition, the PEs will see three more rising edges of PCLK after the stop signal is sent, before PCLK actually stops.

3.4 Controlling the FIFOs

SLAAC1-V defines four input FIFOs (named A0-A3), which the host writes and X0 reads. SLAAC1-V also defines four output FIFOs (named B0-B3) which X0 writes and the host reads. In current distributions, only FIFO_A0, FIFO_A1, FIFO_B0, and FIFO_B1 are actually implemented in hardware. Also, FIFO_A1 and FIFO_B1 are currently driven exclusively by the on-board DMA bus master engine for extremely high throughput. Each FIFO is 64 bits wide. In addition, FIFO_A0 and FIFO_B0 have an extra 4 programmable "tag" bits. The FIFOs are written as follows:

```
// Have to play games like this to get
// 64 bit transfers on a 32-bit CPU
PULONG data[] = {top_half, bottom_half};
s1board->Enqueue(&data, 8 /* bytes */, SLAAC_FIFO_A0);
```

Similarly, the FIFOs are read like this:

```
PULONG data[2];
s1board->Dequeue(&data, 8, /* bytes */, SLAAC_FIFO_B0);
top_half = data[0];
bottom_half = data[1];
```

3.4.1 Low-overhead Enqueue/Dequeue equivalents

(FIFO_A0 / FIFO_B0 only)

Note that the Enqueue method has some overhead, in that it always checks your data pointer and byte count for correct alignment, and it always waits the **full** flag of the FIFO to clear before writing. Dequeue does similar data checking and waits on the **empty** flag before reading the FIFOs. In some cases, you know precisely the FIFO patterns that your

system will use, and will want to access the FIFOs with less overhead. This can be done with the following methods:

```
// Assumes 8 bytes, no alignment check
s1board->WriteFifo(ptr_to_data);
// Assumes 8 bytes, no alignment check,
// no checking full (NB = non-blocking)
s1board->WriteFifoNB(ptr_to_data);
// Same as WriteFifo() except that we only
// transfer 32 bits (1 less bus cycle)
s1board->WriteFifo32(data);
// Same as WriteFifoNB() except that we
// only transfer 32 bits (1 less bus cycle)
s1board->WriteFifo32NB(data);
```

Each of these assumes the "default" input FIFO, SLAAC_FIFO_A0. Similarly, versions of Dequeue exist (*ReadFifo(…)*) which reduce the overhead of reading the default output FIFO, SLAAC_FIFO_B0. Note that these methods are irrelevant to FIFO_A1/FIFOB1 because they are DMA-driven.

3.4.2 Accessing the FIFO tags

(FIFO_A0 / FIFO_B0 only)

FIFO_A0 and FIFO_B0 each have a 4-bit tag that can be written / read with the methods WriteFifoTag / ReadFifoTag, such as:

```
s1board->WriteFifoTag(SLAAC_FIFO_A0, 5);
```

In this case, the tag '5' will be passed with every new FIFO word to X0, until the tag is changed (the tags are actually queued up with the data).

3.5 Accessing the PE Memories

SLAAC1-V gives the host rich control of PE memories. All PE memories are visible to the host (2 for X0, 4 for each of X1 and X2) through the *Read()* and *Write()* API calls. The calls have several overloaded variations, each specifying a slightly different behavior. These variations are summarized below for *Read()*.

Method	Description
`Read(void *buffer, ACS_ADDRESS *addr, int byte_count);`	32-bit preemptive read
`Read(void *buffer, ACS_ADDRESS *addr, size_t word_count, size_t word_size);`	General preemptive read (word_size==4 means 32-bit, word_size==8 means 36-bit)
`ReadNonPreemptive(void *buffer, ACS_ADDRESS *addr, int byte_count);`	32-bit non-preemptive read
`ReadNonPreemptive(void *buffer, ACS_ADDRESS *addr, size_t word_count, size_t word_size);`	General non-preemptive read (word_size==4 means 32-bit, word_size==8 means 36-bit)

The argument and naming conventions for the *Write()* methods are identical. The meaning of these different modes is detailed in the following sections.

3.5.1 32-bit mode vs. 36-bit mode

SLAAC1-V's 36-bit memory word is an awkward fit for the 32-bit PCI bus. Because of this awkward fit, memories can be accessed in either 32-bit mode or 36-bit mode. In 32-bit mode, the host can only see the least significant 32 bits of each memory word; the trade-off is obviously a big win in throughput, as it only takes one PCI cycle to transfer a memory word. Using 36-bit mode allows the host to access the entire memory word at the expense of wasted bandwidth (twice the number of bus cycles in order to get only 12.5% more bits). The general forms of *Read(void *buffer, ACS_ADDRESS *address, size_t word_count, size_t word_size)* and *Write(void *buffer, ACS_ADDRESS *address, size_t word_count, size_t word_size)* allow us to select 32-bit or 36-bit mode as desired. This is shown below.

```
ACS_ADDRESS address;
address.pe = SLAAC_X0;
address.mem = 0;        // We are reading X0M0
address.offset = 10;
s1board->Read(buffer, &address, 1000, 4 /* BYTES PER WORD */);

/* This call would be equivalent to the above: */
s1board->Read(buffer, &address, 4000 /* BYTES */);

unsigned int buffer2[] = {1, 2, 3, 4, 5, 6, 7, 8}
address.pe = SLAAC_X2;
address.mem = 1;        // We are writing X2M1
s1board->Write(buffer, &address, 4, 8 /* BYTES PER WORD */);
```

In this example, we read 1000 words from X0M0 as before, using 32-bit transfers; a second form of this operation is also shown, which uses a byte-count instead of a word-count/word-size; this form is deprecated and is only preserved for backwards-compatibility. The *Write()* call shows how we would write the 36-bit words 0x100000002, 0x300000004, 0x500000006, 700000008 to X2M1, using the 36-bit transfer mode.

3.5.2 Preemptive vs. non-preemptive

A "preemptive" memory access stops the user clock (PCLK) on SLAAC1-V during the access. This is the safest mode of operation and is guaranteed to be completely transparent to the user's circuit. Note that if the host stops the clock in the middle of a PE write to memory, **the address that was being written may appear to the host to have bogus data**. However, when IF releases the bus, the current write operation will complete successfully, and the PEs will not notice the difference. Just realize that one data word may appear bogus when you read it over the external memory bus. Therefore, if your system absolutely depends on reading correct data at every location, it is safest to have some sort of protocol for when the host can read the memories (i.e. a time when you know that the PEs are not writing to memory).

However for very high-throughput applications, it may be necessary to read/write memories without stopping PCLK, so that the user circuit can continue to operate on different memories. In this case, we can use *ReadNonPreemptive()* and *WriteNonPreemptive()* forms of the API. These calls change the bus switches just like *Read()*/*Write()*, but since PCLK is still running the transfer is no longer transparent to the user clock. The memory port that is being accessed by the host is no longer available to X0 until the transfer is over. Please refer to the User VHDL Manual for more details on the memory hierarchy.

3.5.3 DMA vs. PCI slave mode and pipelined vs. flow-through mode

The memory interface can be driven either by PCI slave accesses or by the on-board master-mode DMA controller. Obviously using the DMA controller wherever possible is desirable to gain considerable performance; of course both modes of operation are functionally equivalent. Also, the PE memories have two different speed modes, "pipelined" and "flow-through". Pipelined mode tolerates a higher clock rate at the expense of an extra cycle of read latency.

Both of these memory modes (DMA vs. PCI slave, pipelined vs. flow-through) are controlled by the method *SetMemoryMode(BOOLEAN use_dma, BOOLEAN pipelined)*. The transfer mode is stored in the Slaac1VBoard object and affects all subsequent memory accesses when set. The PE memory mode is controlled by a register in X0 that is broadcast to all memories, so remember to reset the PE memory mode, if necessary, any time that X0 is reconfigured!

3.6 Performing PE Readback

SLAAC1-V supports readback of all PEs. Currently there is no built-in mechanism for associating a bitstream with symbols in a symbol file, although third-party solutions are being developed such as the JHDL debugging environment from BYU. Correct readback depends on setting the right bitgen options when you run the Xilinx tools, and on correctly connecting the VIRTEX_CAPTURE primitive in the user VHDL design. (All of this is done for you in the provided SLAAC1-V VHDL models and makefiles; refer to the SLAAC1-V User VHDL Guide for more details). Raw readback data can be retrieved as follows:

```
int byte_count;
PUCHAR bits = s1board->GetRawReadback(SLAAC_X0, ReadbackCLB,
      &byte_count);
free(bits); // Remember to do this, since the array is malloc'ed
            // by GetRawReadback!
```

Alternatively, the BlockRams can be read by substituting ReadbackBRAM for ReadbackCLB in the above.

3.7 FPGA Control

The Slaac1Board API permits the user to access the global asynchronous reset (GSR), global tristate (GTS), and handshake signals to control the FPGAs. To reset the flip-flops on the PEs, we can do:

```
s1board->Reset(SLAAC_ALL_PES, TRUE);
```

To assert the GTS signal to X1, we could do:

```
s1board->GTS(SLAAC_X1, TRUE);
```

Also, the method GSR is identical to Reset. As explained in the sections on clocking and interrupts, the handshake bus to each PE can be configured to signal a clock-stop or an interrupt. Alternatively, these handshake signals can be configured as generic bi-directional control bits at the disposal of the user. For example, say that we want to drive "01" to the bottom 2 bits of the X2 handshake bus, and then want to read X2's response on the 3rd bit. We can control each bit independently, as follows:

```
s1board->SetHandshake(SLAAC_X2, Tristate, Drive0, Drive1);
```

3.8 Reprogramming the Configuration Controller ("It's not a bug, it's a feature!")

Every once in a while a vicious militant hacker breaks into our systems and discretely inserts errors into our source code (ahem). SLAAC1-V is capable of reprogramming its own boot PROMs with updated bitfiles, allowing you to download the latest fixes, features, etc. from our website and instantly upgrade your configuration controller (a.k.a "XVPI"; this is a fourth FPGA in charge of performing all configuration and readback operations). This is most easily done from the debugger, "slaac1db". From here, you simply invoke the command:

```
<< SLAAC1DB >> eeprom write new_config.bit
```

This process takes a couple of minutes, as the programming clock for the PROMs is 100 KHz. Alternatively, you can perform the upgrade through the API:

```
ACS_CONFIG config;
s1board->LoadConfigurationFile(filename, &config);
s1board->ProgramXVPIProm((PUCHAR) config.bitstream,
config.count);
```

3.9 Interrupt Control

The SLAAC1V API provides for rich control of SLAAC1V-generated interrupts. The board itself has one PCI interrupt signal, with (currently) 10 internal interrupt sources:

- X0
- X1
- X2
- PCLK step done
- PCI counter done

- DMA Error
- DMA A1 done
- DMA B1 done
- DMA MEM0 done
- DMA MEM1 done

Other sources may be implemented in the future. In general, the user has no need to monitor the DMA interrupts as these are all handled internal to the device driver. To synchronize with an interrupt, you need to enable the interrupt and then call the *WaitForInterrupt()* method, as shown below:

```
s1board->EnableInterrupt(SLAAC_INT_X0);
s1board->WaitForInterrupt(SLAAC_INT_X0);
```

Or, even trickier, how about one thread waiting on any one of a number of interrupts:

```
int source = s1board->WaitForInterrupt(SLAAC_INT_X0 |
      SLAAC_INT_X1 | SLAAC_INT_X2);
if (source == SLAAC_INT_X0) { // respond to X0 interrupt }
else if (source == SLAAC_INT_X1) { // respond to X1 interrupt }
else …
```

Furthermore, if you have a fixed response to a given interrupt, you can pass a function pointer as a user-mode ISR that will be invoked every time the interrupt is thrown (assuming your ISR thread goes back to sleep before the next interrupt):

```
void MyISR(int interrupt_status){ … }

void main(…) {
      …
      s1board->SetISR(&MyISR, SLAAC_INT_PCLK_COUNTDOWN);
      …
```

Pretty slick, huh? Probably more than anyone will ever use, but it was fun to make.

Windows NT specific

The SLAAC1-V Windows NT driver sets up individual Windows events which are signaled when a given SLAAC1-V interrupt occurs. You can use these events to set up even more complex synchronization if you want. To wait for the interrupt event, you call the Win API function *WaitForSingleObject()* or *WaitForMultipleObjects()*, as needed. You can obtain the handles to the interrupt events through the Slaac1VBoard method *GetInterruptEvent()*, as shown below:

```
s1board->EnableInterrupt(SLAAC_INT_X0);
HANDLE x0_intr_event = s1board->GetInterruptEvent(SLAAC_INT_X0);
WaitForSingleObject(x0_intr_event, INFINITE);
```

Please refer to the Windows API documentation for more information on how to use Windows thread synchronization functions.

3.10 Slaac1VBoard Register Access

DISCLAIMER: THIS IS FOR ADVANCED USERS ONLY. DIRECTLY PLAYING WITH THE LOW-LEVEL REGISTERS CAN ROYALLY GOOF UP YOUR DESIGN, AND IN THE EXTREME CASE EVEN FRY YOUR BOARD. That being said, it's easy to directly access the SLAAC1-V registers when needed. The most common case when this is used is for accessing the user-definable registers. IF reserves eight specific register addresses for X0 to use as it wishes, as defined in the SLAAC1-V User VHDL Guide.

The best method for accessing the registers is through the SlaacRegister objects contained in the Slaac1VBoard. Each register is represented by a different SlaacRegister. For example, the user-defined registers are contained in Slaac1VBoard::UserRegister[8]. A SlaacRegister has *read()* and *write()* methods for accessing the data. So for example, if you want to check the value in user register 0, and depending on the result write a value to user register 1, you would do:

```
if (s1board->UserRegister[0].read() == 1)
      s1board->UserRegister[1].write(17);
else
      s1board->UserRegister[1].write(23);
```

Furthermore, a SlaacRegister can provide access-safe / type-safe access to individual bitfields in the register. A SlaacRegister can contain a collection of Bitfield objects. A Bitfield describes a sub-range of bits in a register with a static offset and mask value; when you access data via the Bitfields, you are guaranteed not to disturb the other bits in the register. These operations are all inlined, and so should impose little to no overhead versus raw pointer accesses. For example:

```
s1board->XnCtrlReg.write(0); // Clears the whole register
s1board->XnCtrlReg.x0_gsr.write(0); // Clears only the x0_gsr bit
s1board->XnCtrlReg.x0_gsr.assert(); // Same as .write(1), except
                                    // with no shifting /
                                    // masking overhead
ULONG state = s1board->XnCtrlReg.x0_gsr.read();
                                    // Returns either 1 or 0 –
                                    // shifting is taken care of
s1board->XnCtrlReg.x0_gsr = 1;      // Same as .write(1); uses
                                    // operator overloading
ULONG state = s1board->XnCtrlReg.x0_gsr;
                                    // Same as .read()
```

In some instances, you want to write a bunch of bits in parallel, without causing individual bus cycles. This can be easily achieved with the overloaded operators "|", "|=", "&", "&=", "~". For example, say that we want to simultaneously deassert all GSR bits in XnCtrlReg without disturbing the rest of the register:

```
s1board->XnCtrlReg &= ~(s1board->XnCtrlReg.x0_gsr |
                        s1board->XnCtrlReg.x1_gsr |
                        s1board->XnCtrlReg.x2_gsr);
```

In this case, the overloaded '|' operator returns an integer that is the logical-OR of the masks for the three bitfields. Then, the '&=' operator does the following: XnCtrlReg.write(XnCtrlReg.read() & (value)), which is the operation that is naturally expressed.

Last note: for really bare-metal access, you can bypass this SlaacRegister stuff and just get a pointer to the three memory spaces that are mapped by the driver. GetControlSpacePtr() returns the pointer to the control registers, GetIfMemSpacePtr returns the pointer to the PE memory space, and GetDMAFifoSpacePtr returns the pointer to the DMA-controlled FIFO space. From these pointers you can directly access the memories, although you have to turn the memory bus around before you can access the PE memories (ask before trying this). Also, you can use the register offset values in Slaac1VBoard.h as indices into the control-space pointer to directly read / write raw registers. This method is not recommended.